Exhibit 4

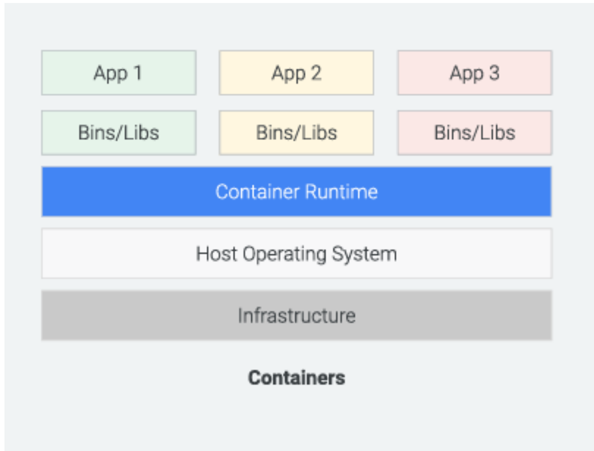
U.S. Patent No. 7,784,058 ("'058 Patent")

Accused Instrumentalities: Google products and services using secure containerized applications, including without limitation Google Kubernetes Engine, Cloud Run, and Migrate to Containers, and all versions and variations thereof since the issuance of the asserted patent.

Each Accused Instrumentality infringes the claims in substantially the same way, and the evidence shown in this chart is similarly applicable to each Accused Instrumentality. Each claim limitation is literally infringed by each Accused Instrumentality. However, to the extent any claim limitation is not met literally, it is nonetheless met under the doctrine of equivalents because the differences between the claim limitation and each Accused Instrumentality would be insubstantial, and each Accused Instrumentality performs substantially the same function, in substantially the same way, to achieve the same result as the claimed invention. Notably, Defendant has not yet articulated which, if any, particular claim limitations it believes are not met by the Accused Instrumentalities.

Claim 1

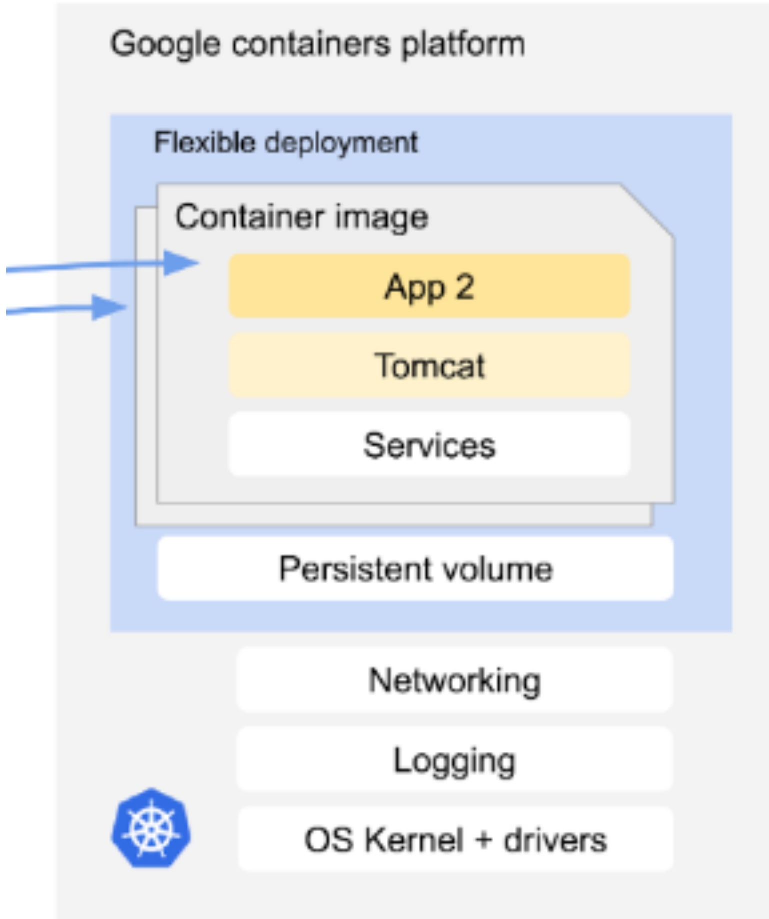
Claim 1	Accused Instrumentalities
[1pre] 1. A computing system for executing a plurality of software applications comprising:	To the extent the preamble is limiting, each Accused Instrumentality comprises or constitutes a computing system for executing a plurality of software applications as claimed. *See* claim limitations below. *See also, e.g.*: Google Kubernetes Engine (GKE) clusters provide secured and managed Kubernetes services with autoscaling and multi-cluster support. GKE lets you deploy, manage, and scale containerized applications on Kubernetes, powered by Google Cloud. https://cloud.google.com/migrate/containers/docs/getting-started Use Migrate to Containers to modernize traditional applications away from virtual machine (VM) instances and into native containers that run on Google Kubernetes Engine (GKE), GKE Enterprise clusters, or Cloud Run platform. You can migrate workloads from VMs that run on VMware or Compute Engine, giving you the flexibility to containerize your existing workloads with ease. Migrate to Containers supports modernization of IBM WebSphere, JBoss, Apache, Tomcat, WordPress, Windows IIS applications, as well as containerisation of Linux-based applications. https://cloud.google.com/migrate/containers/docs/getting-started.

Claim 1	Accused Instrumentalities
	A container is a way of packaging a given application's code and dependencies so that the application will run easily in any computing environment. This solves the common problem of  https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 1	Accused Instrumentalities
	Containers can run virtually anywhere, greatly easing development and deployment: on Linux, Windows, and Mac operating systems; on virtual machines or on physical servers; on a developer's machine or in data centers on-premises; and of course, in the public cloud. Containers are lightweight packages of your application code together with dependencies such as specific versions of programming language runtimes and libraries required to run your software services. https://cloud.google.com/learn/what-are-containers
[1a] a) a processor;	Each Accused Instrumentality comprises a processor. *See, e.g.*:

Claim 1	Accused Instrumentalities
	Containers virtualize CPU, memory, storage, and network resources at the operating system level, providing developers with a view of the OS logically isolated from other applications. https://cloud.google.com/learn/what-are-containers • **Higher utilization and density**, leveraging automatic bin-packing and auto-scaling capabilities, Kubernetes places containers optimally in nodes based on required resources while scaling as needed, without impairing availability. In addition, unlike VMs, all containers on a single node share one copy of the operating system and don't each require their own OS image and vCPU, resulting in a much smaller memory footprint and CPU needs. This means more workloads running on fewer compute resources. https://cloud.google.com/blog/products/containers-kubernetes/how-migrate-for-anthos-improves-vm-to-container-migration Containers use specific features of the Linux kernel that "trick" individual applications into thinking they're in their own unique environment, even though multiple applications share the same host kernel. (If you're not familiar with the Linux kernel, it's a part of the operating system that communicates between processes--requests that do user tasks like opening a file, running a program-- and the hardware. It manages resources like memory and CPU to meet these requests). https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 1	Accused Instrumentalities
[1b] b) an operating system having an operating system kernel having OS critical system elements (OSCSEs) for running in kernel mode using said processor; and,	Each Accused Instrumentality comprises an operating system having an operating system kernel having OS critical system elements (OSCSEs) for running in kernel mode using said processor. *See, e.g.*: • Containers are much more lightweight than VMs • Containers virtualize at the OS level while VMs virtualize at the hardware level • Containers share the OS kernel and use a fraction of the memory VMs require https://cloud.google.com/learn/what-are-containers **Kernel mode** Kernel mode refers to the processor mode that enables software to have full and unrestricted access to the system and its resources. The OS kernel and kernel drivers, such as the file system driver, are loaded into protected memory space and operate in this highly privileged kernel mode. https://www.techtarget.com/searchdatacenter/definition/kernel

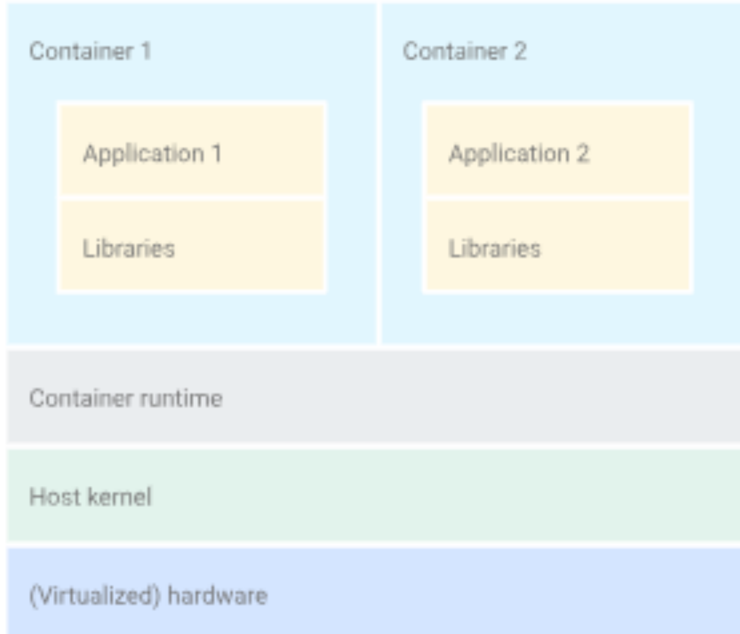
Claim 1	Accused Instrumentalities
	 https://cloud.google.com/blog/products/application-modernization/shift-your-apps-to-container-based-workloads-on-the-command-line

Claim 1	Accused Instrumentalities
	The migration prerequisites are dependent on your specific migration environment. Confirm that your workloads' OS and source platform are compatible for migration by reviewing the prerequisites for your specific migration environment: https://cloud.google.com/migrate/containers/docs/setting-up-overview Containers use specific features of the Linux kernel that "trick" individual applications into thinking they're in their own unique environment, even though multiple applications share the same host kernel. (If you're not familiar with the Linux kernel, it's a part of the operating system that communicates between processes--requests that do user tasks like opening a file, running a program-- and the hardware. It manages resources like memory and CPU to meet these requests). https://services.google.com/fh/files/misc/why_container_security_matters.pdf The **GNU C Library**, commonly known as **glibc**, is the GNU Project implementation of the C standard library. It is a wrapper around the system calls of the Linux kernel for application use. Despite its name, it now also directly supports C++ (and, indirectly, other programming languages). It was started in the 1980s by the Free Software Foundation (FSF) for the GNU operating system. https://en.wikipedia.org/wiki/Glibc
[1c] c) a shared library having shared library critical system elements (SLCSEs) stored therein for use by the plurality of software applications in user mode and	Each Accused Instrumentality comprises a shared library having shared library critical system elements (SLCSEs) stored therein for use by the plurality of software applications in user mode. *See, e.g.*:

Claim 1	Accused Instrumentalities
	A "container image" is your application and its dependencies, and uses a "base image" as the basis for the container image The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities. https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 1	Accused Instrumentalities
	A base image is the starting point for most container-based development workflows. Developers start with a base image and layer on top of it the necessary libraries, binaries, and configuration files used to run their application. Many base images are basic or minimal Linux distributions: Debian, Ubuntu, Red Hat Enterprise Linux (RHEL), Rocky Linux, or Alpine. Developers can consume these images directly from Docker Hub or other sources. There are official providers along with a wide variety of other downstream repackagers that layer software to meet customer needs. Google maintains base images for building its own applications. These images are built from the same source that Docker Hub uses. Therefore, they match the images you would get from Docker Hub. https://cloud.google.com/software-supply-chain-security/docs/base-images The preconfigured base images provided by Cloud Workstations contain only a minimal environment with IDE, basic Linux terminal and language tools and a `sshd` server. To expedite the environment setup of specific development use cases, you can create custom container images that extend these base images to pre-install tools and dependencies and that run automation scripts. For custom container images, we recommend setting up a pipeline to automatically rebuild these images when the Cloud Workstations base image is updated, in addition to running a container scanning tool such as Artifact Analysis to inspect any additional dependencies you added. You're responsible for maintaining and updating custom packages and dependencies added to custom images. https://cloud.google.com/workstations/docs/customize-container-images

Claim 1	Accused Instrumentalities
	A container is a way of packaging a given application's code and dependencies so that the application will run easily in any computing environment. This solves the common problem of Containers solve the portability problem by isolating the application and its dependencies so they can be moved seamlessly between machines. A process running in a container lives isolated from the underlying environment. You control what it can see and what resources it can access. This helps you use resources more efficiently and not worry about the underlying infrastructure. One of the primary reasons to adopt containers is for your applications to be decoupled from the underlying environment and support higher resource utilization by "bin packing" multiple workloads onto each server. As such, the architecture of containers means that they're deployed with multiple containers sharing the same kernel. The core components of the Linux kernel that are used for containers are **cgroups** — control groups, which define the resources like CPU and memory which are available to a given process — and **namespaces**, which are a way of separating processes by restricting what each process can see, so that system resources "appear" isolated to the process. https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 1	Accused Instrumentalities
	 https://cloud.google.com/architecture/best-practices-for-operating-containers For example, Migrate to Containers automatically generates a container image, a Dockerfile for day-2 image updates and application revisions, Kubernetes deployment YAMLs and (where relevant) a persistent data volume onto which the application data files and persistent state are copied. This automated, intelligent extraction is https://cloud.google.com/blog/products/containers-kubernetes/how-migrate-for-anthos-improves-vm-to-container-migration

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>
Containers are lightweight packages of your application code together with dependencies such as specific versions of programming language runtimes and libraries required to run your software services.

https://cloud.google.com/learn/what-are-containers

About storage drivers

To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way.

Storage drivers versus Docker volumes

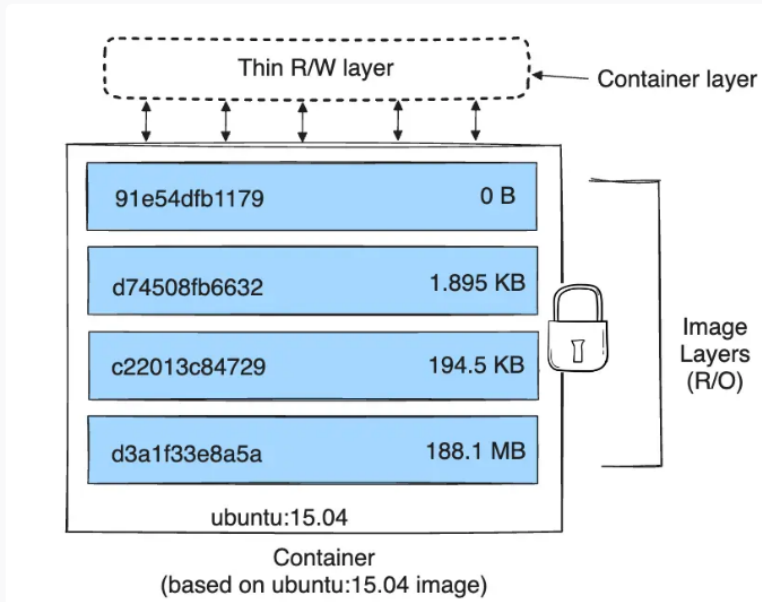
Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer.

Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance.

https://docs.docker.com/storage/storagedriver/
</td></tr>
</table>

Claim 1	Accused Instrumentalities

	# Images and layers
	
	A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:
	
	`# syntax=docker/dockerfile:1`
	
	`FROM ubuntu:22.04`
	`LABEL org.opencontainers.image.authors="org@example.com"`
	`COPY . /app`
	`RUN make /app`
	`RUN rm -r $HOME/.cache`
	`CMD python /app/app.py`
	
	This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.
	
	https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	# Volumes Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts: https://kubernetes.io/docs/concepts/storage/volumes/ ## Container environment The Kubernetes Container environment provides several important resources to Containers: • A filesystem, which is a combination of an image and one or more volumes. • Information about the Container itself. • Information about other objects in the cluster. https://kubernetes.io/docs/concepts/containers/container-environment/

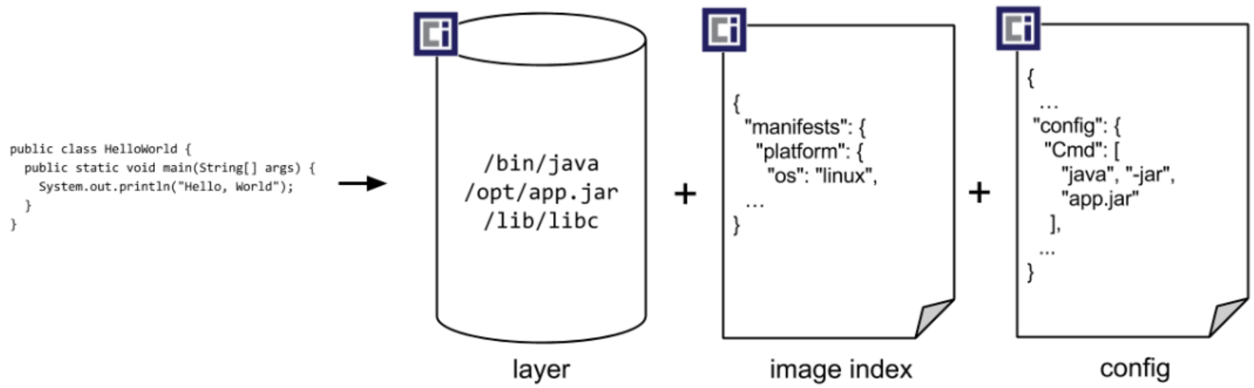
Claim 1	Accused Instrumentalities
	Images A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/ **Volumes** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	# Open Container Initiative ## Image Format Specification This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Overview

At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.



https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md
</td></tr>
</table>

Claim 1	Accused Instrumentalities
	OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>
<h2>Layer</h2>

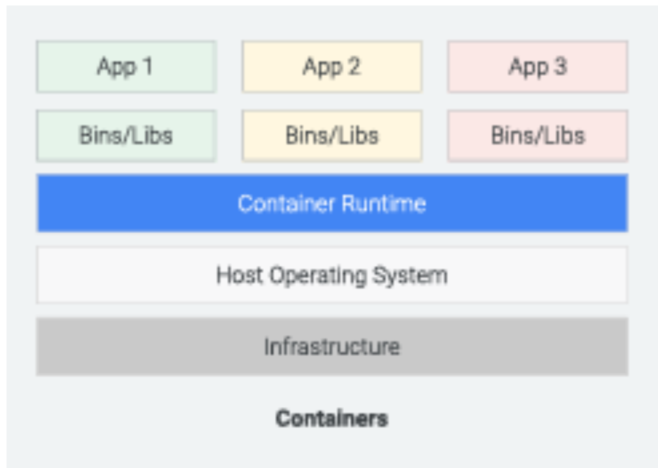
Image filesystems are composed of layers.
Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer.
Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer.
Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem.

<h2>Image JSON</h2>

Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes.
The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers.
This JSON is considered to be immutable, because changing it would change the computed ImageID.
Changing it means creating a new derived image, instead of changing the existing image.

https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md
</td></tr>
<tr><td>[1d] i) wherein some of the SLCSEs stored in the shared library are functional replicas of OSCSEs and are accessible</td><td>In each Accused Instrumentality, some of the SLCSEs stored in the shared library are functional replicas of OSCSEs and are accessible to some of the plurality of software applications and when</td></tr>
</table>

Claim 1	Accused Instrumentalities
to some of the plurality of software applications and when one of the SLCSEs is accessed by one or more of the plurality of software applications it forms a part of the one or more of the plurality of software applications,	one of the SLCSEs is accessed by one or more of the plurality of software applications it forms a part of the one or more of the plurality of software applications. For example, a base image serves as a self-contained unit that encompasses all the necessary components for an application to run, including the application code, runtime environment, system tools, and dependencies (i.e., SLCSEs). The images are based on existing Linux distributions, such as Debian and Ubuntu, including essential system elements (i.e., functional replicas of OSCSEs). Each container image is based on a specific base image, which contains the application code, and dependencies, including system libraries or shared library critical system elements (SLCSEs). When the container runs the image, it creates a runtime instance of that container image. *See, e.g.*: Many base images are basic or minimal Linux distributions: Debian, Ubuntu, Red Hat Enterprise Linux (RHEL), Rocky Linux, or Alpine. Developers can consume these images directly from Docker Hub or other sources. There are official providers along with a wide variety of other downstream repackagers that layer software to meet customer needs. https://cloud.google.com/software-supply-chain-security/docs/base-images A container is a way of packaging a given application's code and dependencies so that the application will run easily in any computing environment. This solves the common problem of

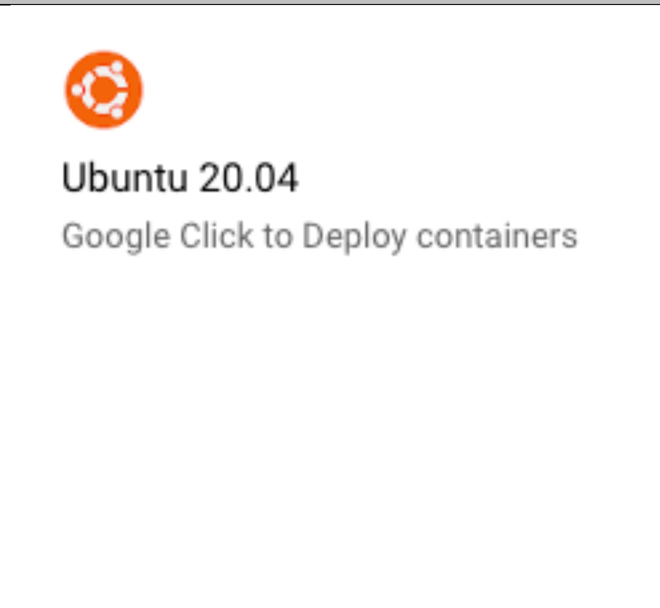
Claim 1	Accused Instrumentalities
	A "container image" is your application and its dependencies, and uses a "base image" as the basis for the container image 

Claim 1	Accused Instrumentalities
	The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities. https://services.google.com/fh/files/misc/why_container_security_matters.pdf

OS	Repository path	Google Cloud Marketplace listing
Debian 10 "Buster"	`marketplace.gcr.io/google/debian10`	Google Cloud Marketplace
Debian 11 "Bullseye"	`marketplace.gcr.io/google/debian11`	Google Cloud Marketplace
Debian 12 "Bookworm"	`marketplace.gcr.io/google/debian12`	Google Cloud Marketplace
Rocky Linux 8	`marketplace.gcr.io/google/rockylinux8`	Google Cloud Marketplace
Rocky Linux 9	`marketplace.gcr.io/google/rockylinux9`	Google Cloud Marketplace
Ubuntu 20.04	`marketplace.gcr.io/google/ubuntu2004`	Google Cloud Marketplace
Ubuntu 22.04	`marketplace.gcr.io/google/ubuntu2204`	Google Cloud Marketplace

https://cloud.google.com/software-supply-chain-security/docs/base-images

Claim 1	Accused Instrumentalities
	Debian 10 "Buster" Google Click to Deploy containers Open source OS

Claim 1	Accused Instrumentalities
	 https://console.cloud.google.com/marketplace/browse?filter=solution-type:container

Claim 1	Accused Instrumentalities
	About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. **Storage drivers versus Docker volumes** Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Images and layers

A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

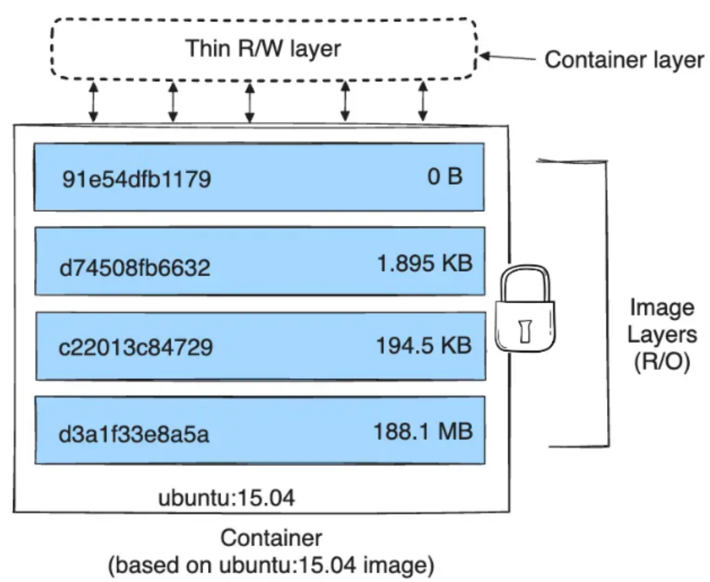
```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
LABEL org.opencontainers.image.authors="org@example.com"
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.

https://docs.docker.com/storage/storagedriver/

</td></tr>
</table>

Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  https://docs.docker.com/storage/storagedriver/

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Volumes

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:

https://kubernetes.io/docs/concepts/storage/volumes/

Container environment

The Kubernetes Container environment provides several important resources to Containers:

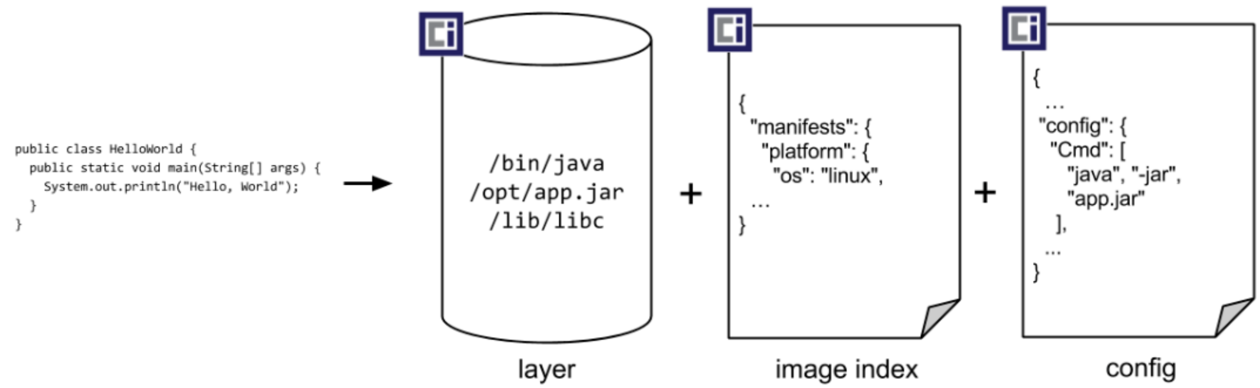
- A filesystem, which is a combination of an image and one or more volumes.
- Information about the Container itself.
- Information about other objects in the cluster.

https://kubernetes.io/docs/concepts/containers/container-environment/

</td></tr>
</table>

Claim 1	Accused Instrumentalities
	Images A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/ **Volumes** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	# Open Container Initiative ## Image Format Specification This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 1	Accused Instrumentalities
	Overview At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.  https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 1	Accused Instrumentalities
	OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>
<h3>Layer</h3>

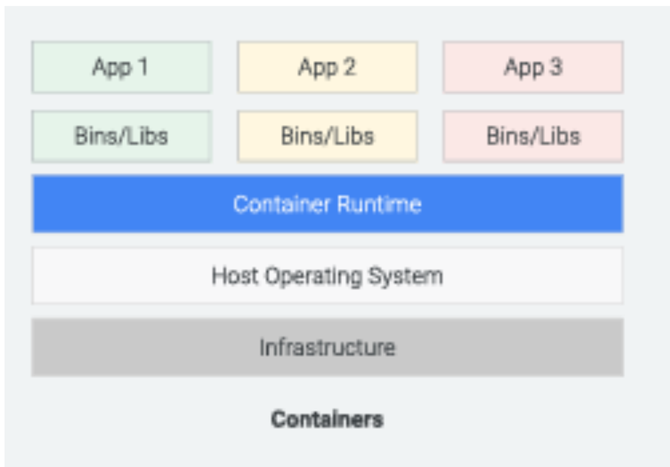
Image filesystems are composed of layers.
Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer.
Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer.
Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem.

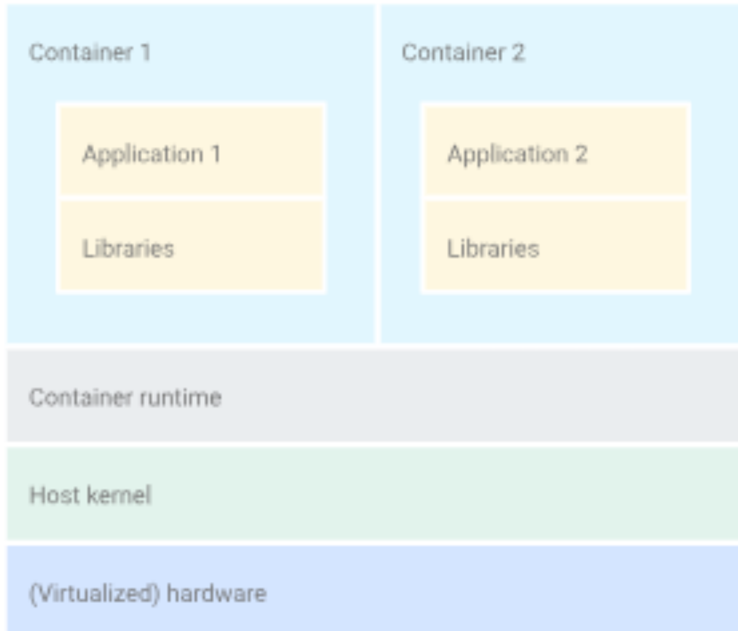
<h3>Image JSON</h3>

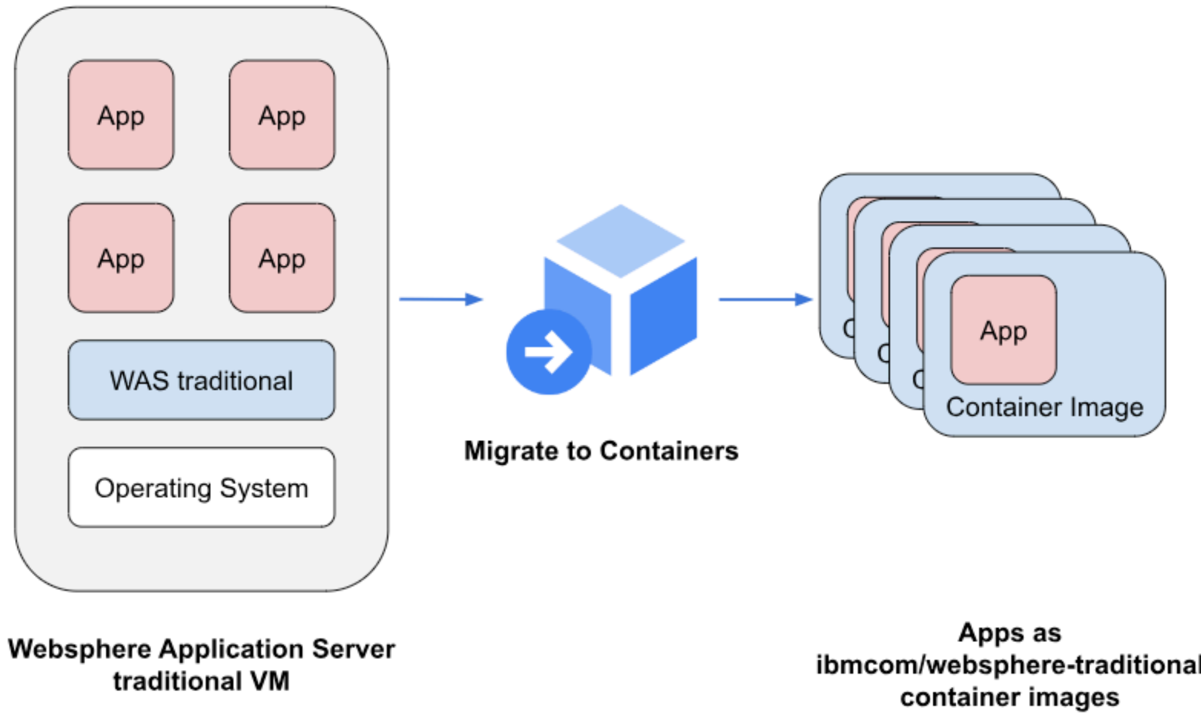
Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes.
The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers.
This JSON is considered to be immutable, because changing it would change the computed ImageID.
Changing it means creating a new derived image, instead of changing the existing image.

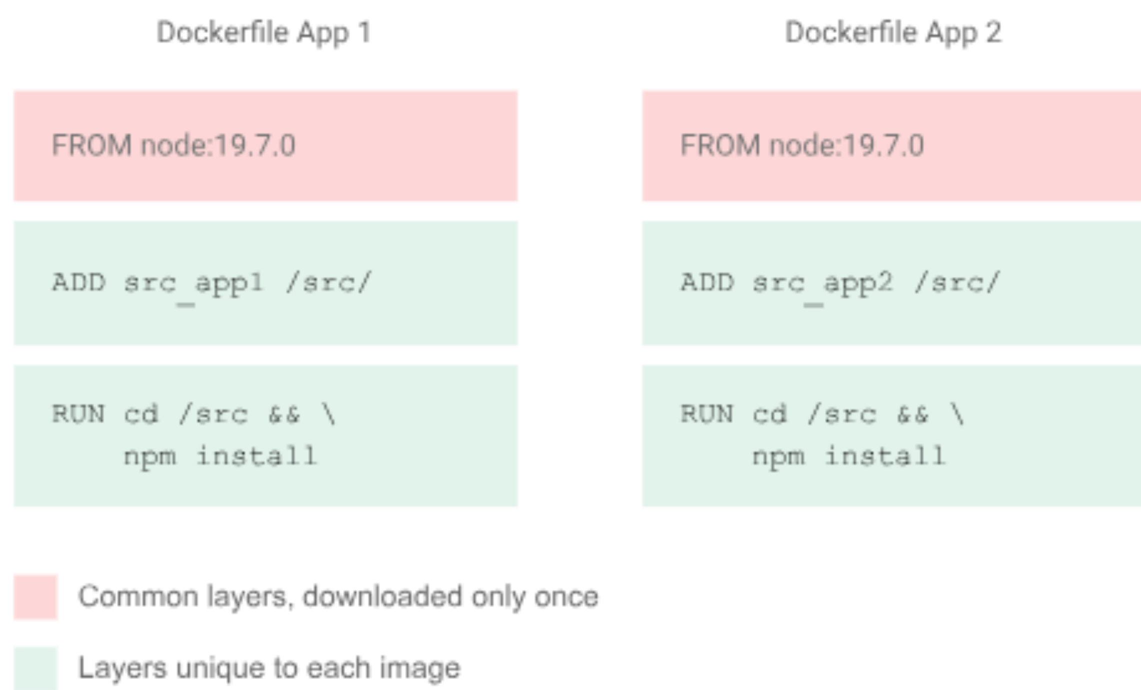
https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md
</td></tr>
<tr><td>[1e] ii) wherein an instance of a SLCSE provided to at least a first of the plurality of software applications from the</td><td>In each Accused Instrumentality, an instance of a SLCSE provided to at least a first of the plurality of software applications from the shared library is run in a context of said at least first of the plurality of software applications without being shared with other of the plurality of software applications and where at least a second of the plurality of software applications running under the</td></tr>
</table>

Claim 1	Accused Instrumentalities
shared library is run in a context of said at least first of the plurality of software applications without being shared with other of the plurality of software applications and where at least a second of the plurality of software applications running under the operating system have use of a unique instance of a corresponding critical system element for performing same function, and	operating system have use of a unique instance of a corresponding critical system element for performing same function. When a Docker or Kubernetes image is used to create a container, it creates a separate and isolated instance of a runtime environment which is independent of other containers running on the same host. Each container has its own instance of base images and its own data. The containers run in isolation, ensuring that the SLCSEs stored in the shared library are accessible to the software applications running in their respective containers. The image includes essential system files, libraries, and dependencies required to run the software application within the container. The containers can share common dependencies and components using layered images. This means that multiple containers utilize the same base image to create an instance. When an instance of SLCSE is provided from the base image (i.e., from the shared library) to an individual container including application software, it operates in isolation and runs its own instance of the software application without sharing resources or critical system elements with other containers. This ensures that each container has its own isolated context. Docker or Kubernetes containers can share common dependencies and components using layered images. This means that multiple containers can utilize the same base image. Therefore, each container, containing the application software running under the operating system, utilizes a unique instance of the corresponding critical system element to execute the respective application software for performing a same or a different function. *See, e.g.*: A container is a way of packaging a given application's code and dependencies so that the application will run easily in any computing environment. This solves the common problem of https://services.google.com/fh/files/misc/why_container_security_matters.pdf

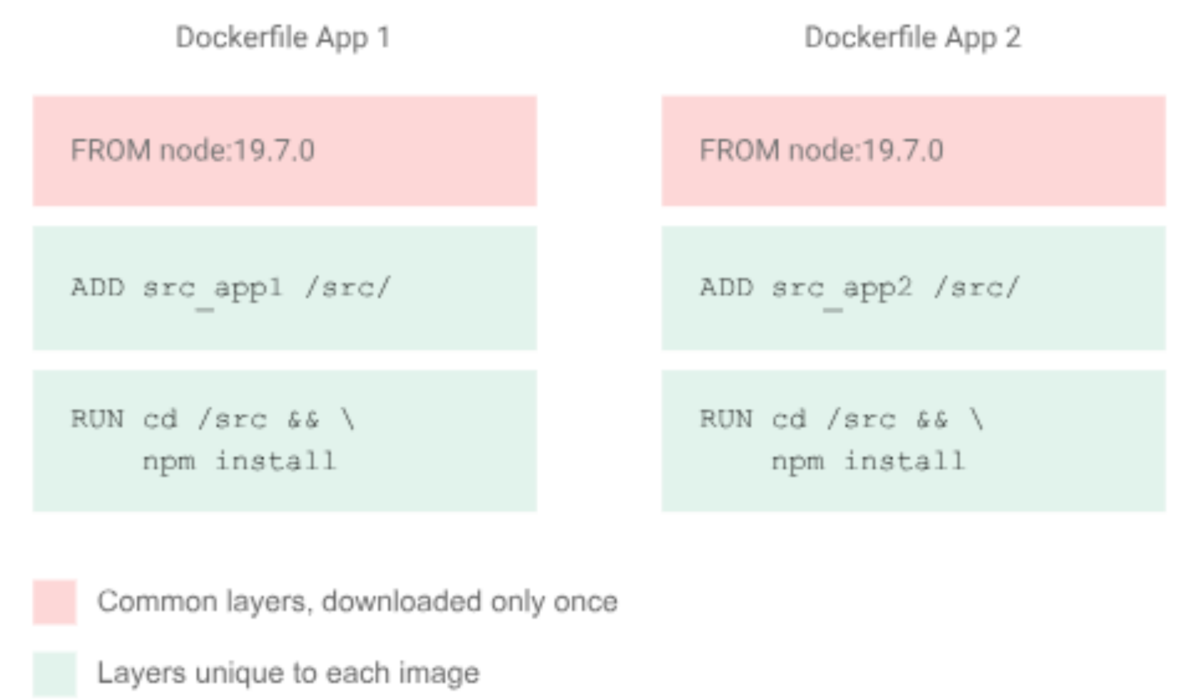
Claim 1	Accused Instrumentalities
	Containers solve the portability problem by isolating the application and its dependencies so they can be moved seamlessly between machines. A process running in a container lives isolated from the underlying environment. You control what it can see and what resources it can access. This helps you use resources more efficiently and not worry about the underlying infrastructure. The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities.  https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 1	Accused Instrumentalities
	 https://cloud.google.com/architecture/best-practices-for-operating-containers

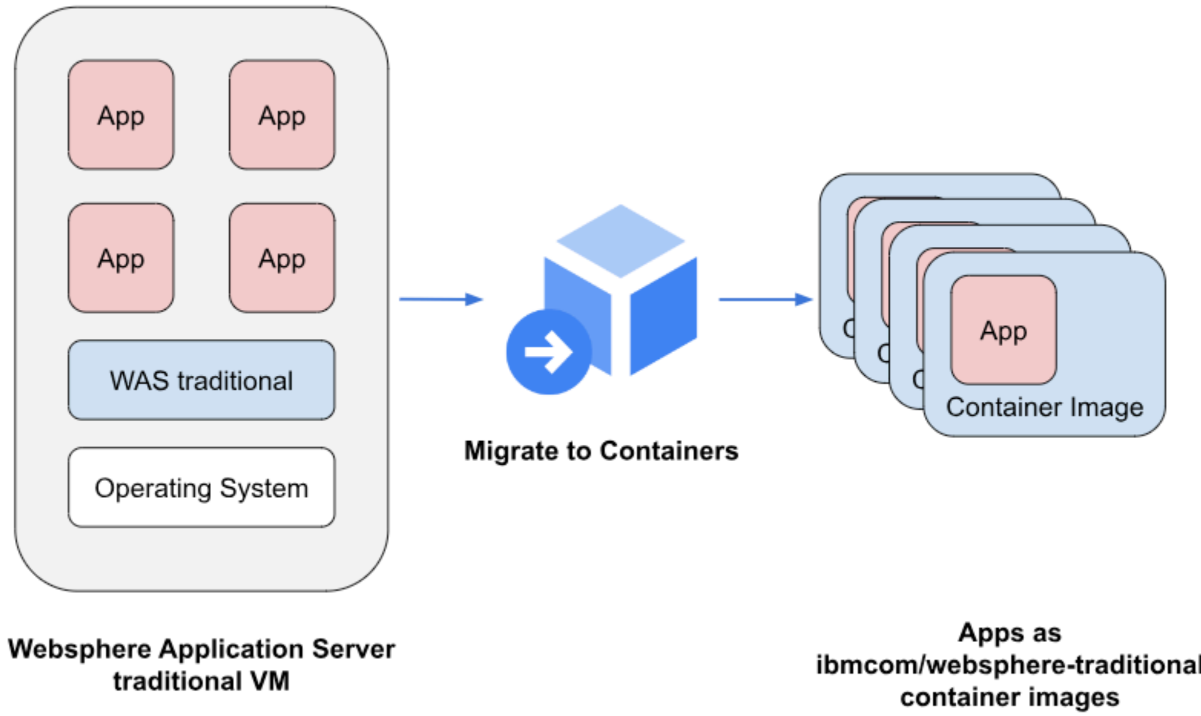
Claim 1	Accused Instrumentalities
	 https://cloud.google.com/migrate/containers/docs/migrating-overview

Claim 1	Accused Instrumentalities
	 https://cloud.google.com/architecture/best-practices-for-building-containers One method of packaging an application into a container is with the use of a Dockerfile. The Dockerfile is similar to a script which instructs the daemon on how to assemble the container image. See the Dockerfile reference documentation) for more information. Using the Dockerfile method to build a container requires direct knowledge about the application in order to assemble the container. The first step to creating a Dockerfile is selecting an image that will be used as the basis of your image.This image should be a parent or base image maintained and published by a trusted source, usually your company. https://codelabs.developers.google.com/developing-containers-with-dockerfiles#2

Claim 1	Accused Instrumentalities
[1f] iii) wherein a SLCSE related to a predetermined function is provided to the first of the plurality of software applications for running a first instance of the SLCSE, and wherein a SLCSE for performing a same function is provided to the second of the plurality of software applications for running a second instance of the SLCSE simultaneously.	In each Accused Instrumentality, a SLCSE related to a predetermined function is provided to the first of the plurality of software applications for running a first instance of the SLCSE, and wherein a SLCSE for performing a same function is provided to the second of the plurality of software applications for running a second instance of the SLCSE simultaneously. For example, in Docker or Kubernetes containers, each container operates independently, and a base image includes essential system files, libraries, and dependencies (i.e., SLCSEs) required to run the software application within the container. Based on information and belief, each element, such as system files, libraries, and dependencies (i.e., SLCSE) is associated with an execution of a predetermined function related to the application. When an image is used to create a container in the Accused Instrumentality, an instance of the SLCSE is provided to a software application. Therefore, different instances of the SLCSE are provided to different applications for performing either a same or a different function, simultaneously. *See, e.g.*: Containers solve the portability problem by isolating the application and its dependencies so they can be moved seamlessly between machines. A process running in a container lives isolated from the underlying environment. You control what it can see and what resources it can access. This helps you use resources more efficiently and not worry about the underlying infrastructure. The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities. https://services.google.com/fh/files/misc/why_container_security_matters.pdf

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>


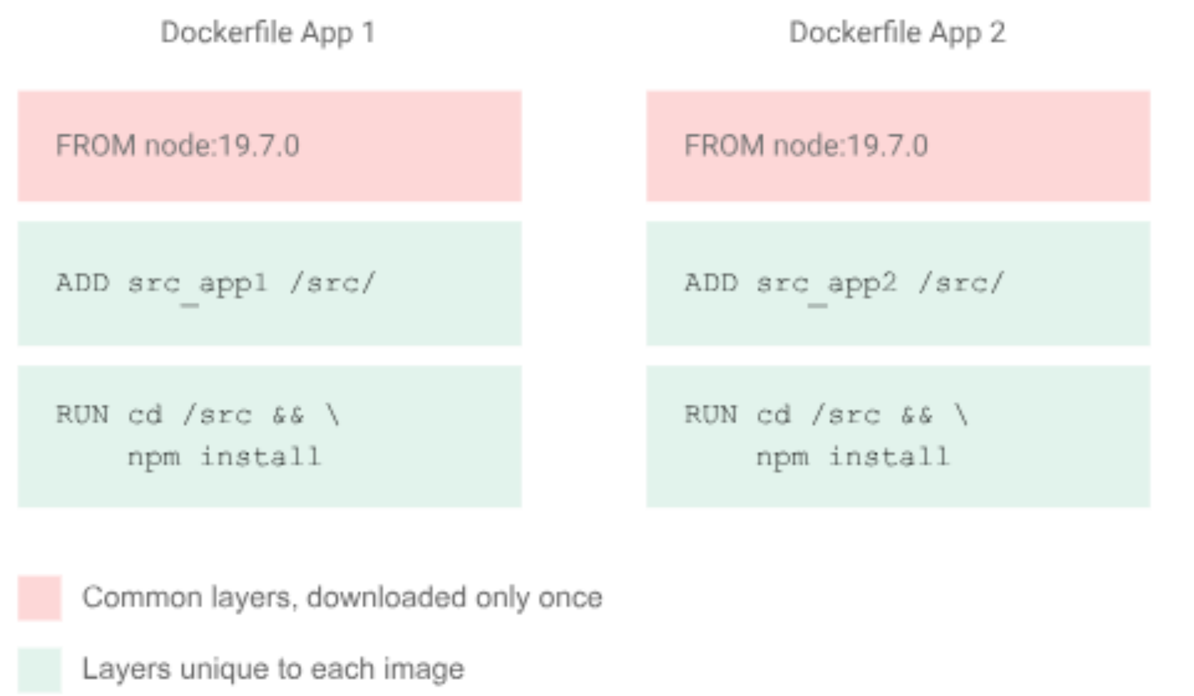
https://cloud.google.com/architecture/best-practices-for-building-containers
</td></tr>
</table>

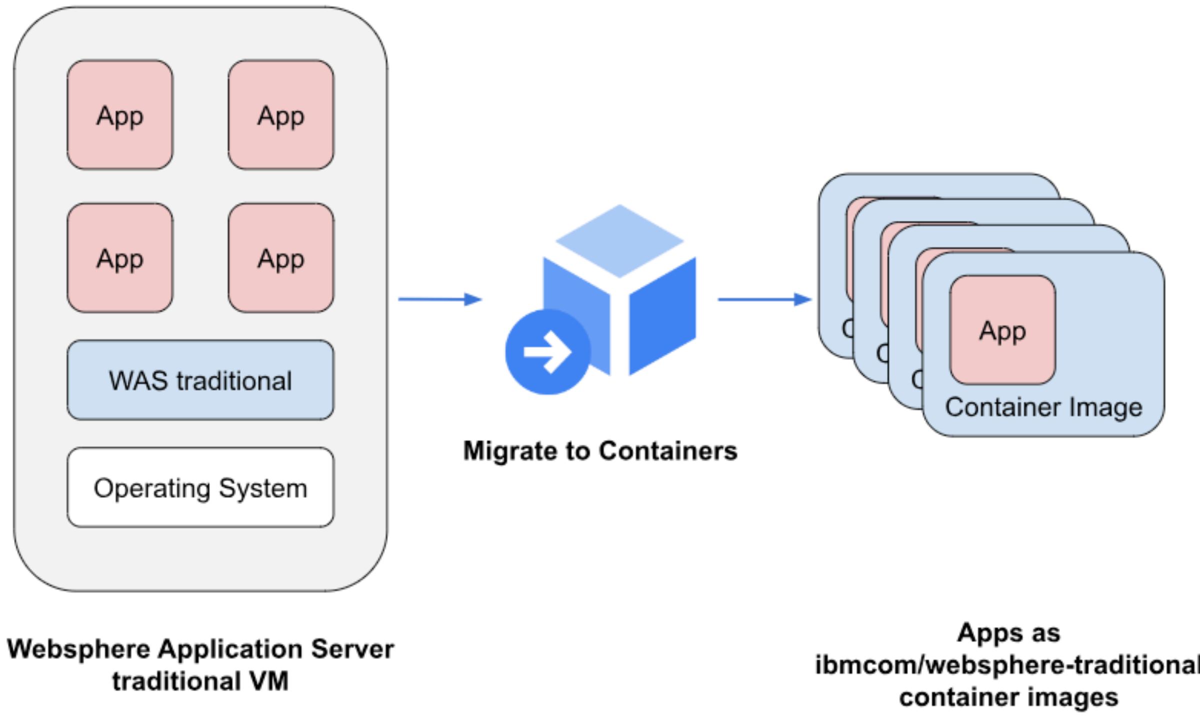
Claim 1	Accused Instrumentalities
	 https://cloud.google.com/migrate/containers/docs/migrating-overview

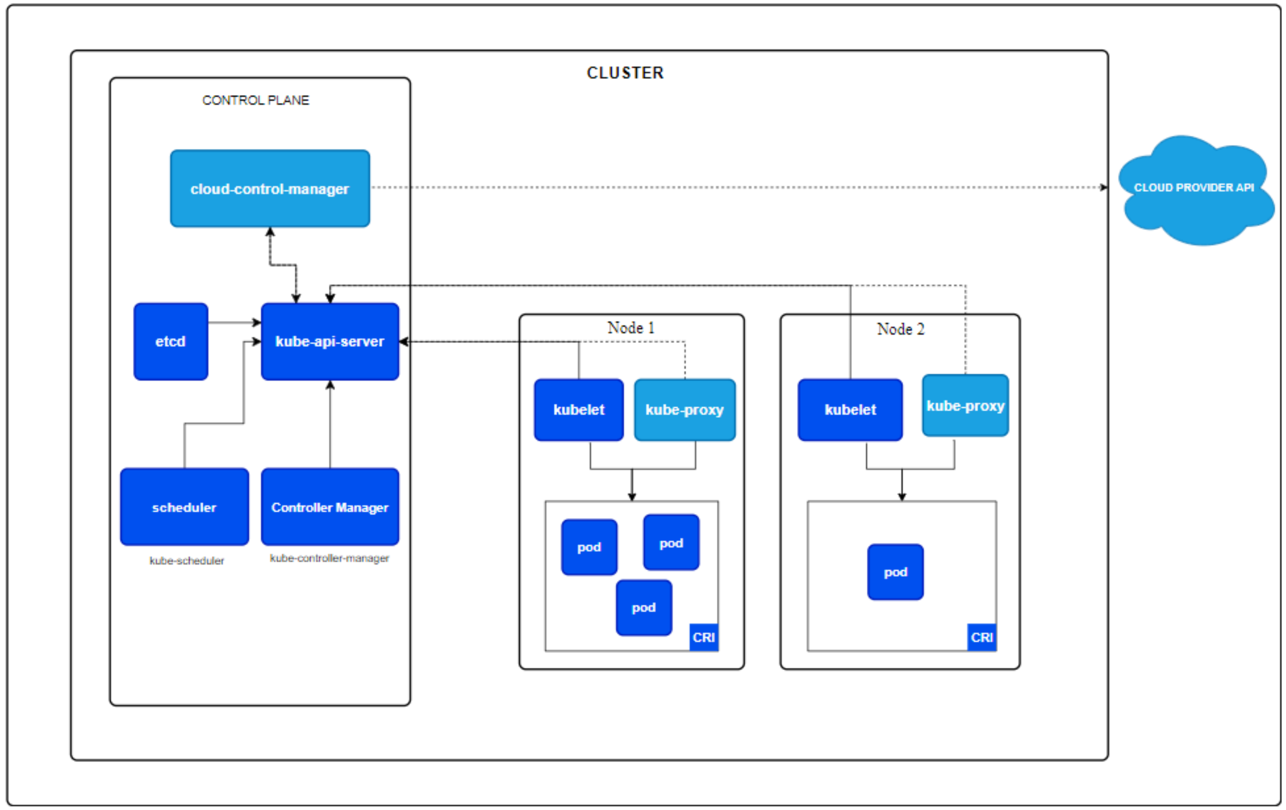
Claim 2

Claim 2	Accused Instrumentalities
2. A computing system as defined in claim 1, wherein in operation, multiple instances of an SLCSE stored in the shared library run	Each Accused Instrumentality comprises or constitutes a computing system as defined in claim 1, wherein in operation, multiple instances of an SLCSE stored in the shared library run simultaneously within the operating system.

Claim 2	Accused Instrumentalities
simultaneously within the operating system.	For example, an individual host/node runs multiple containers and/or pods simultaneously, each of which has an instance of an SLCSE. *See, e.g.*: Containers solve the portability problem by isolating the application and its dependencies so they can be moved seamlessly between machines. A process running in a container lives isolated from the underlying environment. You control what it can see and what resources it can access. This helps you use resources more efficiently and not worry about the underlying infrastructure. The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities. https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 2	Accused Instrumentalities
	 https://cloud.google.com/architecture/best-practices-for-building-containers

Claim 2	Accused Instrumentalities
	 https://cloud.google.com/migrate/containers/docs/migrating-overview

Claim 2	Accused Instrumentalities
	 Kubernetes cluster architecture https://kubernetes.io/docs/concepts/architecture/

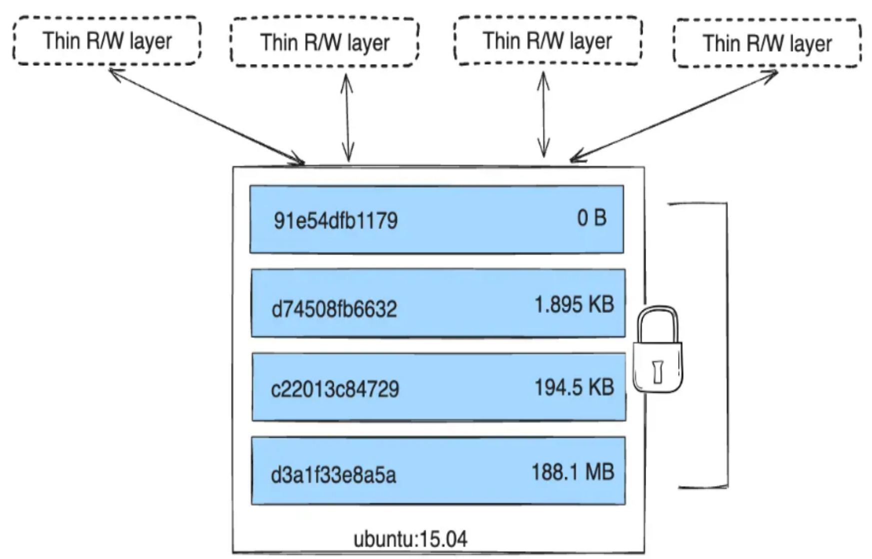
Claim 2	Accused Instrumentalities
	Containers Each container that you run is repeatable; the standardization from having dependencies included means that you get the same behavior wherever you run it. Containers decouple applications from the underlying host infrastructure. This makes deployment easier in different cloud or OS environments. Each node in a Kubernetes cluster runs the containers that form the Pods assigned to that node. Containers in a Pod are co-located and co-scheduled to run on the same node. https://kubernetes.io/docs/concepts/containers/

<table>
<tr><th>Claim 2</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>
<h1>Kubernetes Scheduler</h1>
In Kubernetes, scheduling refers to making sure that Pods are matched to Nodes so that Kubelet can run them.
<h2>Scheduling overview</h2>
A scheduler watches for newly created Pods that have no Node assigned. For every Pod that the scheduler discovers, the scheduler becomes responsible for finding the best Node for that Pod to run on. The scheduler reaches this placement decision taking into account the scheduling principles described below.
If you want to understand why Pods are placed onto a particular Node, or if you're planning to implement a custom scheduler yourself, this page will help you learn about scheduling.
https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/
</td></tr>
</table>

Claim 2	Accused Instrumentalities
	Running containers Docker runs processes in isolated containers. A container is a process which runs on a host. The host may be local or remote. When you execute `docker run`, the container process that runs is isolated in that it has its own file system, its own networking, and its own isolated process tree separate from the host. https://docs.docker.com/engine/reference/run/

Claim 3

Claim 3	Accused Instrumentalities
3. A computing system according to claim 1 wherein OSCSEs corresponding to and capable of performing the same function as SLCSEs remain in the operating system kernel.	Each Accused Instrumentality comprises or constitutes a computing system according to claim 1 wherein OSCSEs corresponding to and capable of performing the same function as SLCSEs remain in the operating system kernel. For example, both Docker and Kubernetes systems preserve the host kernel substantially unchanged; therefore the OSCSEs corresponding to the SLCSEs remain in the operating system kernel. *See, e.g.*: Most base images are basic or minimal Linux distributions: Debian, Ubuntu, Redhat, Centos, or Alpine. Developers usually consume these images directly from Docker Hub, or other sources. There are official providers along with a wide variety of other downstream repackagers that layer software to meet customer needs. https://cloud.google.com/software-supply-chain-security/docs/base-images

Claim 3	Accused Instrumentalities
	Container image files are complete, static and executable versions of an application or service and differ from one technology to another. Docker images are made up of multiple layers, which start with a base image that includes all of the dependencies needed to execute code in a container. Each image has a readable/writable layer on top of static unchanging layers. Because each container has its own specific container layer that customizes that specific container, underlying image layers can be saved and reused in multiple containers. An Open Container Initiative (OCI) https://www.techtarget.com/searchitoperations/definition/container-containerization-or-container-based-virtualization Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.  https://docs.docker.com/storage/storagedriver/

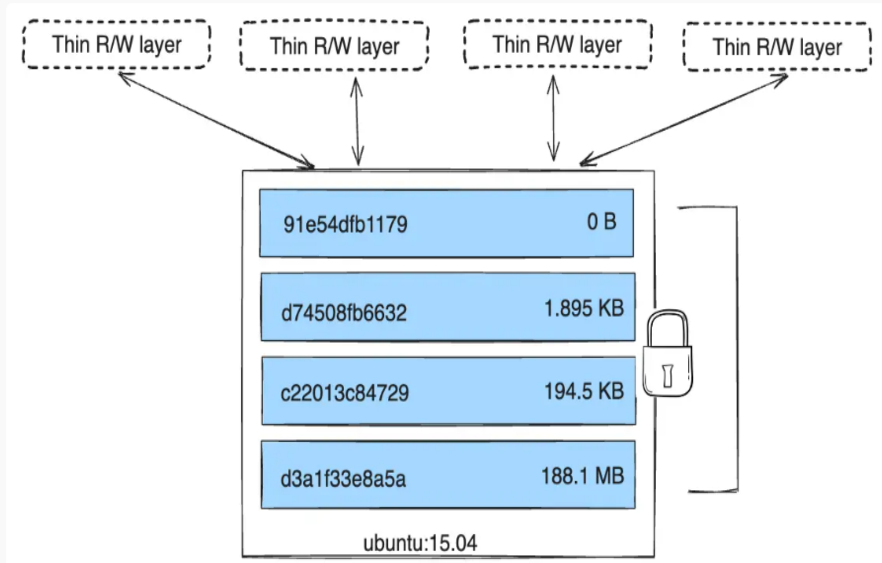
Claim 4

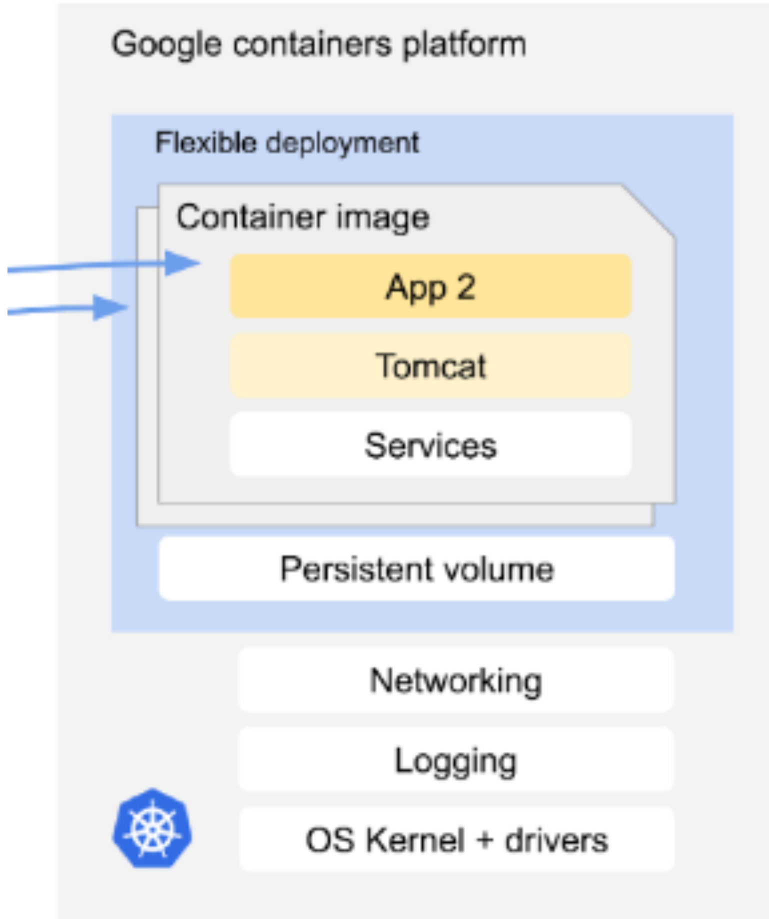
Claim 4	Accused Instrumentalities
4. A computing system according to claim 1 wherein the one or more SLCSEs provided to one of the plurality of software applications having exclusive use thereof, use system calls to access services in the operating system kernel.	Each Accused Instrumentality comprises or constitutes a computing system according to claim 1 wherein the one or more SLCSEs provided to one of the plurality of software applications having exclusive use thereof, use system calls to access services in the operating system kernel. For example, the SLCSEs in a container use system calls to access services in the operating system kernel. For example, the glibc library (or other similar library) in the container uses system calls to interface with the host Linux kernel. In general, system calls can be observed using a tool such as strace. *See, e.g.*: The **GNU C Library**, commonly known as **glibc**, is the GNU Project implementation of the C standard library. It is a wrapper around the system calls of the Linux kernel for application use. Despite its name, it now also directly supports C++ (and, indirectly, other programming languages). It was started in the 1980s by the Free Software Foundation (FSF) for the GNU operating system. https://en.wikipedia.org/wiki/Glibc

	We can now get the process id directly from the cgroup. It will be located in the `cgroup.procs` file.
	
	`### Terminal 2 - Worker Node ###`
	
	`# Get the process id`
	`$ cat cri-containerd-ceeeef06afe89c8223d33b11e8d9e0b207118ac4dac3af826687668ee1ee`
	`16254`
	
	`# Validate what is running under the process`
	`$ ps aux | grep 16254`
	`azureus+    16254 0.0  0.1 713972 10476 ?        Ssl  15:04   0:00 ./faultyapp`
	`azureus+   94806  0.0  0.0   7004  2168 pts/0    S+   16:22   0:00 grep --color=a`
	
	Got it! With that, we can try to find out what is going out inside the app. Lets try to run `strace` to get some more insight.
	
	`### Terminal 2 - Worker Node ###`
	
	`$ sudo strace -p 16254 -f`
	`...`
	`# The app is trying to read a file port.txt`
	`[pid 16269] openat(AT_FDCWD, "port.txt", O_RDONLY|O_CLOEXEC <unfinished ...>`
	`[pid 16254] epoll_pwait(5,  <unfinished ...>`
	`# The file does not exist`
	`[pid 16269] <... openat resumed>)       = -1 ENOENT (No such file or directory)`
	`[pid 16254] <... epoll_pwait resumed>[], 128, 0, NULL, 0) = 0`
	`[pid 16269] write(1, "Something went wrong...\\n", 24 <unfinished ...>`
	
	After filtering the output, we can see the application is trying to read a text file called `port.txt`, and a few lines later, there is a message stating `ENOENT (No such file or directory)`. Let's create that file.
	https://www.berops.com/blog/a-different-method-to-debug-kubernetes-pods

<u>Claim 5</u>

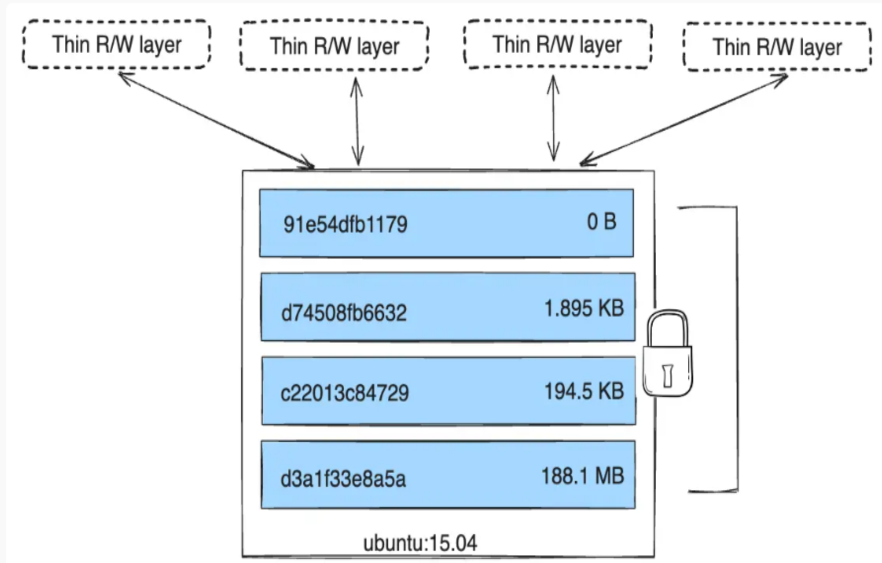
Claim 5	Accused Instrumentalities
5. A computing system according to claim 1 wherein the operating system kernel comprises a kernel module adapted to serve as an interface between an SLCSE in the context of an application program and a device driver.	Each Accused Instrumentality comprises or constitutes a computing system according to claim 1 wherein the operating system kernel comprises a kernel module adapted to serve as an interface between an SLCSE in the context of an application program and a device driver. For example, the server (node) includes an operating system having a kernel. The kernel comprises a kernel module which enables applications (including their libraries) to have access to system resources such as storage, *i.e.*, acts as an interface between applications/libraries and OS libraries or drivers *See, e.g.*: **Container images** A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/ Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. ttps://www.techtarget.com/searchitoperations/definition/Docker-image

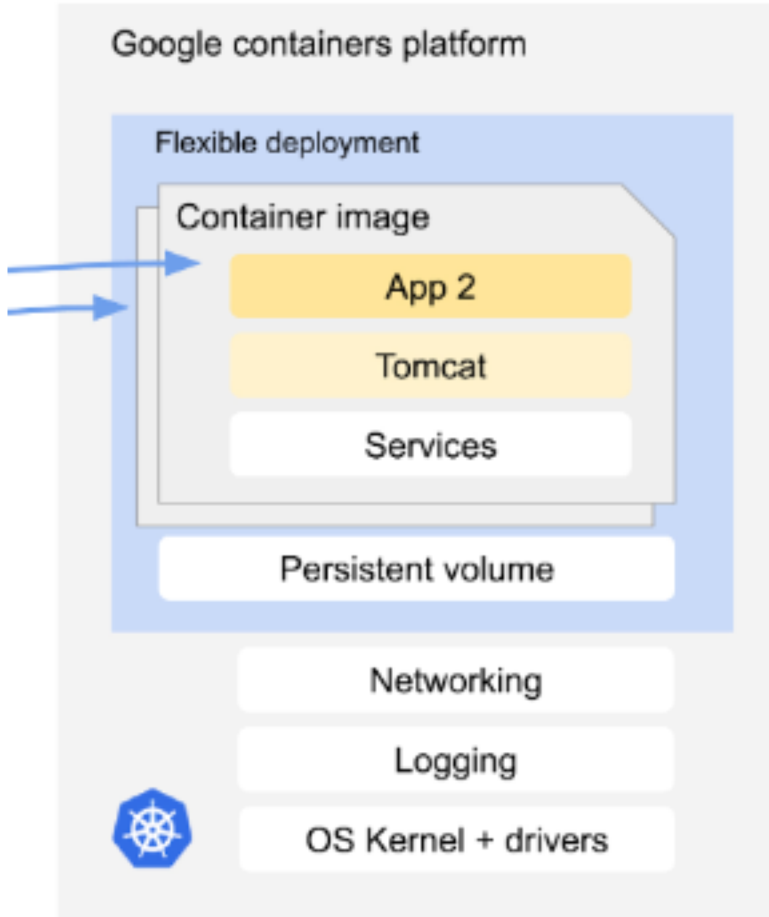
Claim 5	Accused Instrumentalities
	Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.  https://docs.docker.com/storage/storagedriver/ Containers use specific features of the Linux kernel that "trick" individual applications into thinking they're in their own unique environment, even though multiple applications share the same host kernel. (If you're not familiar with the Linux kernel, it's a part of the operating system that communicates between processes--requests that do user tasks like opening a file, running a program-- and the hardware. It manages resources like memory and CPU to meet these requests). https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 5	Accused Instrumentalities
	 https://cloud.google.com/blog/products/application-modernization/shift-your-apps-to-container-based-workloads-on-the-command-line

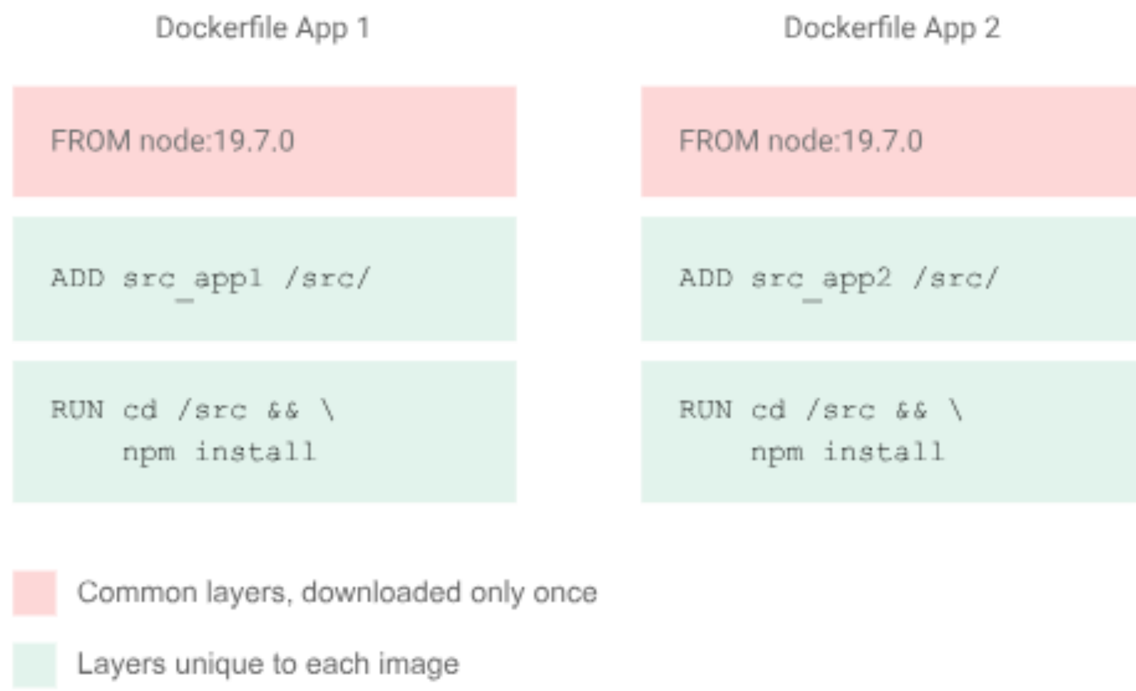
Claim 10

Claim 10	Accused Instrumentalities
10. A computing system according to claim 2 wherein SLCSEs stored in the shared library are linked to particular software applications of the plurality of software applications as the particular software applications are loaded such that the particular software applications have a link that provides unique access to a unique instance of a CSE.	Each Accused Instrumentality comprises or constitutes a computing system according to claim 2 wherein SLCSEs stored in the shared library are linked to particular software applications of the plurality of software applications as the particular software applications are loaded such that the particular software applications have a link that provides unique access to a unique instance of a CSE. For example, the containers can share common dependencies and components using layered images, and multiple containers can use the same base image. Therefore, each container, containing the application software running under the operating system of the server hosting GKE, uses a unique instance of the corresponding critical system element to execute the respective application software and has a link to that unique instance. *See, e.g.*: Container images A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/ Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. ttps://www.techtarget.com/searchitoperations/definition/Docker-image

Claim 10	Accused Instrumentalities
	Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.  https://docs.docker.com/storage/storagedriver/ Containers use specific features of the Linux kernel that "trick" individual applications into thinking they're in their own unique environment, even though multiple applications share the same host kernel. (If you're not familiar with the Linux kernel, it's a part of the operating system that communicates between processes--requests that do user tasks like opening a file, running a program-- and the hardware. It manages resources like memory and CPU to meet these requests). https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 10	Accused Instrumentalities
	 https://cloud.google.com/blog/products/application-modernization/shift-your-apps-to-container-based-workloads-on-the-command-line

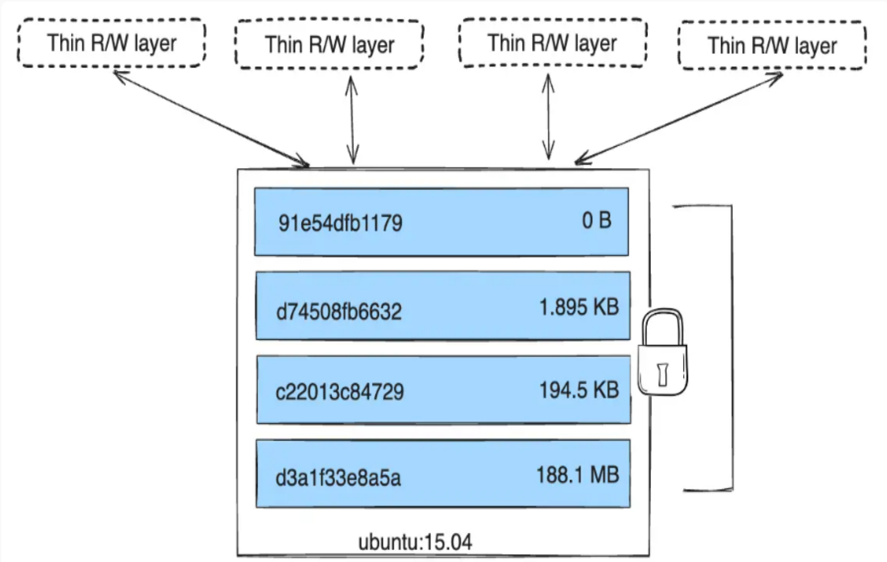
Claim 10	Accused Instrumentalities
	Containers solve the portability problem by isolating the application and its dependencies so they can be moved seamlessly between machines. A process running in a container lives isolated from the underlying environment. You control what it can see and what resources it can access. This helps you use resources more efficiently and not worry about the underlying infrastructure. The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities. https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 10	Accused Instrumentalities
	 https://cloud.google.com/architecture/best-practices-for-building-containers

Claim 18

Claim 18	Accused Instrumentalities
18. A computer system as defined in claim 2 wherein SLCSEs are not copies of OSCSEs.	Each Accused Instrumentality comprises or constitutes a computer system as defined in claim 2 wherein SLCSEs are not copies of OSCSEs. For example, in a typical case the SLCSEs come from a Linux distribution independent of the host operating system, and thus are not identical to the OSCSEs. For another example, the SLCSEs are

Claim 18	Accused Instrumentalities
	provided to the computer system through a separate process than the process by which the OSCSEs are provided to the computer system, and thus are not copied from the OSCSEs. *See, e.g.*: **Container images** A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/ Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. ttps://www.techtarget.com/searchitoperations/definition/Docker-image

Claim 18	Accused Instrumentalities
	Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.  https://docs.docker.com/storage/storagedriver/